Generics means **parameterized types**. The idea is to allow type (Integer, String, … etc., and user-defined types) to be a parameter to methods, classes, and interfaces. Using Generics, it is possible to create classes that work with different data types. An entity such as class, interface, or method that operates on a parameterized type is a generic entity.

Why Generics?

The **Object** is the superclass of all other classes, and Object reference can refer to any object. These features lack type safety. Generics add that type of safety feature. We will discuss that type of safety feature in later examples.
Generics in Java are similar to templates in C++. For example, classes like HashSet, ArrayList, HashMap, etc., use generics very well

Types of Java Generics

Generic Method: Generic Java method takes a parameter and returns some value after performing a task. It is exactly like a normal function, however, a generic method has type parameters that are cited by actual type. This allows the generic method to be used in a more general way. The compiler takes care of the type of safety which enables programmers to code easily since they do not have to perform long, individual type castings.
Generic Classes: A generic class is implemented exactly like a non-generic class. The only difference is that it contains a type parameter section. There can be more than one type of parameter, separated by a

comma. The classes, which accept one or more parameters, are known as parameterized classes or parameterized types.

Generic Class

Like C++, we use <> to specify parameter types in generic class creation. To create objects of a generic class, we use the following syntax.

// To create an instance of generic class

BaseType <Type> obj = new BaseType <Type>()

Note: *In Parameter type we can not use primitives like ‘int’,’char’ or ‘double’.*

```
// Java program to show working of user defined

// Generic classes


// We use < > to specify Parameter type

class Test<T> {

    // An object of type T is declared

    T obj;

    Test(T obj) { this.obj = obj; } // constructor
    public T getObject() { return this.obj; }
}

// Driver class to test above
class Main {
    public static void main(String[] args)
    {
        // instance of Integer type
        Test<Integer> iObj = new Test<Integer>(15);
        System.out.println(iObj.getObject());

        // instance of String type
        Test<String> sObj
            = new Test<String>("Good morning");
        System.out.println(sObj.getObject());
    }
}
```

```
// Java program to show multiple
// type parameters in Java Generics

// We use < > to specify Parameter type
class Test<T, U>
{
    T obj1; // An object of type T
    U obj2; // An object of type U

    // constructor
    Test(T obj1, U obj2)
    {
        this.obj1 = obj1;
        this.obj2 = obj2;
    }

    // To print objects of T and U
    public void print()
    {
        System.out.println(obj1);
        System.out.println(obj2);
    }
}
```

```
// Driver class to test above
class Main
{
        public static void main (String[] args)
        {
                Test <String, Integer> obj =
                        new Test<String, Integer>("Good
morning", 15);

                obj.print();
        }
}
```

Generic Functions:

We can also write generic functions that can be called with different types of arguments based on the type of arguments passed to the generic method. The compiler handles each method.

```
// Java program to show working of user defined
// Generic functions
```

```
class Test {
        // A Generic method example
        static <T> void genericDisplay(T element)
        {

        System.out.println(element.getClass().getName()
                                        + " = " + element);
        }

        // Driver method
        public static void main(String[] args)
        {
                // Calling generic method with Integer argument
                genericDisplay(11);

                // Calling generic method with String argument
                genericDisplay("Hello");

                // Calling generic method with double argument
                genericDisplay(1.0);
        }
}
```

```java
public class GenericMethodTest {
   // generic method printArray
   public static < E > void printArray( E[] inputArray ) {
      // Display array elements
      for(E element : inputArray) {
         System.out.println(element);
      }
      System.out.println();
   }

   public static void main(String args[]) {
      // Create arrays of Integer, Double and Character
      Integer[] intArray = { 1, 2, 3, 4, 5 };
      Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
      Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

      System.out.println("Array integerArray contains:");
      printArray(intArray);   // pass an Integer array

      System.out.println("\nArray doubleArray contains:");
      printArray(doubleArray);   // pass a Double array

      System.out.println("\nArray characterArray
contains:");
      printArray(charArray);   // pass a Character array
   }
}
```

Type Parameters in Java Generics

The type parameters naming conventions are important to learn generics thoroughly. The common type parameters are as follows:

- T – Type
- E – Element
- K – Key
- N – Number
- V – Value

Advantages of Generics:

Programs that use Generics has got many benefits over non-generic code.

1. Code Reuse: We can write a method/class/interface once and use it for any type we want.
2. Type Safety: Generics make errors to appear compile time than at run time (It’s always better to know problems in your code at compile time rather than making your code fail at run time). Suppose you want to create an ArrayList that store name of students, and if by mistake the programmer adds an integer object instead of a string, the compiler allows it. But, when we retrieve this data from ArrayList, it causes problems at runtime.
3. Individual Type Casting is not needed:

4. Generics Promotes Code Reusability: With the help of generics in Java, we can write code that will work with different types of data. For example,
public <T> void genericsMethod (T data) {...}

Here, we have created a generics method. This same method can be used to perform operations on integer data, string data, and so on.

5. Implementing Generic Algorithms: By using generics, we can implement algorithms that work on different types of objects, and at the same, they are type-safe too.